TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 05/10 pp39-50
Volume 19, Number 1, February 2014

Dache: A Data Aware Caching for Big-Data Applications Using the MapReduce Framework

Yaxiong Zhao*, Jie Wu, and Cong Liu

Abstract: The buzz-word *big-data* refers to the large-scale distributed data processing applications that operate on exceptionally large amounts of data. Google's *MapReduce* and Apache's *Hadoop*, its open-source implementation, are the defacto software systems for big-data applications. An observation of the MapReduce framework is that the framework generates a large amount of intermediate data. Such abundant information is thrown away after the tasks finish, because MapReduce is unable to utilize them. In this paper, we propose *Dache*, a data-aware cache framework for big-data applications. In Dache, tasks submit their intermediate results to the cache manager. A task queries the cache manager before executing the actual computing work. A novel cache description scheme and a cache request and reply protocol are designed. We implement Dache by extending Hadoop. Testbed experiment results demonstrate that Dache significantly improves the completion time of MapReduce jobs.

Key words: big-data; MapReduce; Hadoop; caching

1 Introduction

Google MapReduce[1] is a programming model and a software framework for large-scale distributed computing on large amounts of data. Figure 1 illustrates the high-level work flow of a MapReduce job. Application developers specify the computation in terms of a map and a reduce function, and the underlying MapReduce job scheduling system automatically parallelizes the computation across a cluster of machines. MapReduce gains popularity for its simple programming interface and excellent performance when implementing a large spectrum of applications. Since most such applications take a large amount of input data, they are nicknamed "Big-data applications". As shown in Fig. 1, input data is first split and then feed to workers in the map phase. Individual data items are called *records*. The MapReduce system parses the input splits to each worker and produces records. After the map phase, intermediate results generated in the map phase are shuffled and sorted by the MapReduce system and are then fed into the workers in the reduce phase. Final results are computed by multiple reducers and written to the disk.

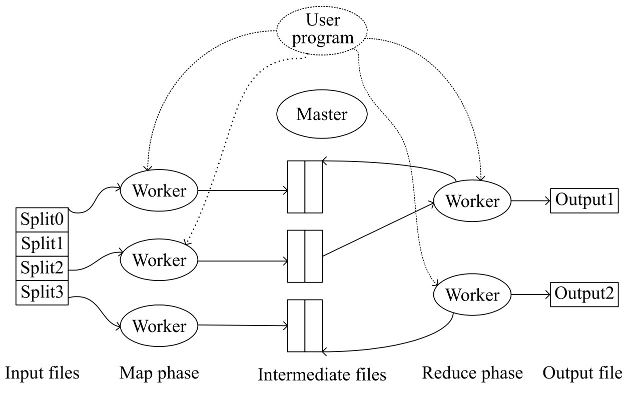


Fig. 1 A high-level illustration of the MapReduce programming model and the underlying implementation architecture.

• Yaxiong Zhao is with Google Inc., Mountain View, CA 94043, USA. This work was done while the author was with Temple University, Philadelphia, PA 19122, USA. E-mail: yaxiongzhao@google.com.
• Jie Wu is with Temple University, Philadelphia, PA 19122, USA. E-mail: jiewu@temple.edu.
• Cong Liu is with Sun Yat-Sen University, Guangzhou 510275, China. E-mail: gzcong@gmail.com.
∗ To whom correspondence should be addressed.
Manuscript received: 2013-12-13; accepted: 2013-12-20

Hadoop[2] is an open-source implementation of the Google MapReduce programming model. Hadoop consists of the Hadoop Common, which provides access to the file systems supported by Hadoop. Particularly, the Hadoop Distributed File System (HDFS) provides distributed file storage and is optimized for large immutable blobs of data. A small Hadoop cluster will include a single master and multiple worker nodes. The master node runs multiple processes, including a JobTracker and a NameNode. The JobTracker is responsible for managing running jobs in the Hadoop cluster. The NameNode, on the other hand, manages the HDFS. The JobTracker and the NameNode are usually collocated on the same physical machine. Other servers in the cluster run a TaskTracker and a DataNode processes. A MapReduce job is divided into tasks. Tasks are managed by the TaskTracker. The TaskTrackers and the DataNode are collated on the same servers to provide data locality in computation.

MapReduce provides a standardized framework for implementing large-scale distributed computation, namely, the big-data applications. However, there is a limitation of the system, i.e., the inefficiency in incremental processing. Incremental processing refers to the applications that incrementally grow the input data and continuously apply computations on the input in order to generate output. There are potential duplicate computations being performed in this process. However, MapReduce does not have the mechanism to identify such duplicate computations and accelerate job execution. Motivated by this observation, we propose Dache, a data-aware cache system for big-data applications using the MapReduce framework. Dache aims at extending the MapReduce framework and provisioning a cache layer for efficiently identifying and accessing cache items in a MapReduce job. The following technical challenges need to be addressed before implementing this proposal.

- **Cache description scheme**. Data-aware caching requires each data object to be indexed by its content. In the context of big-data applications, this means that the cache description scheme needs to describe the application framework and the data contents. Although most big-data applications run on standardized platforms, their individual tasks perform completely different operations and generate different intermediate results. The cache description scheme should provide a customizable indexing that enables the applications to describe their operations and the content of their generated partial results. This is a non-trivial task. In the context of Hadoop, we utilize the sterilization capability provided by the Java[3] language to identify the object that is used by the MapReduce system to process the input data.
- **Cache request and reply protocol**. The size of the aggregated intermediate data can be very large. When such data is requested by other worker nodes, determining how to transport this data becomes complex. Usually the programs are moved to data nodes in order to run the processing locally. However, this may not always be applicable since the identities of the worker nodes may not be easily changed. Data locality is another concern. The protocol should be able to collate cache items with the worker processes potentially that need the data, so that the transmission delay and overhead are minimized.

In this paper, we present a novel cache description scheme. A high-level description is presented in Fig. 2. This scheme identifies the source input from which a cache item is obtained, and the operations applied on the input, so that a cache item produced by the workers in the map phase is indexed properly. In the reduce phase, we devise a mechanism to take into consideration the partition operations applied on the output in the map phase. We also present a method for reducers to utilize the cached results in the map phase to accelerate the execution of the MapReduce job. We implement Dache in the Hadoop project by extending the relevant components. Our implementation follows a non-intrusive approach, so it only requires minimum

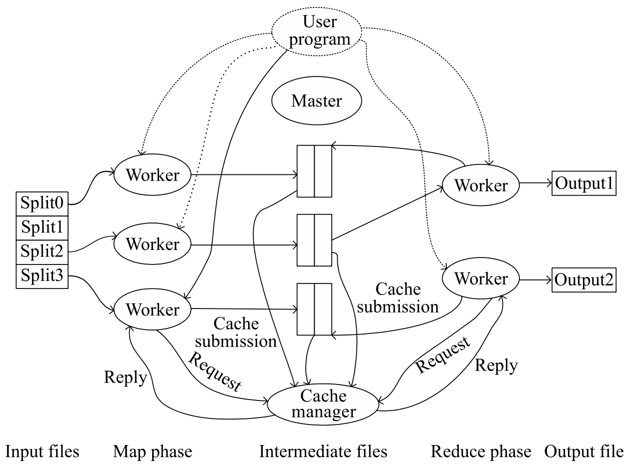


Fig. 2 High-level description of the architecture of Dache. A cache query phase is appended in the map and reduce phases. A *cache manager* is incorporated to manage cache items and answer queries for mappers and reducers.

changes to the application code.

2 Cache Description

2.1 Map phase cache description scheme

Cache refers to the intermediate data that is produced by worker nodes/processes during the execution of a MapReduce task. A piece of cached data is stored in a *Distributed File System* (DFS). The content of a cache item is described by the original data and the operations applied. Formally, a cache item is described by a 2-tuple: {*Origin*, *Operation*}. Origin is the name of a file in the DFS. Operation is a linear list of available operations performed on the Origin file. For example, in the word count application, each mapper node/process emits a list of {*word*, *count*} tuples that record the count of each word in the file that the mapper processes. Dache stores this list to a file. This file becomes a cache item. Given an original input data file, *word_list_08012012.txt*, the cache item is described by {*word_list_08012012.txt, item count*}. Here, *item* refers to white-space-separated character strings. Note that the new line character is also considered as one of the white spaces, so item precisely captures the word in a text file and item count directly corresponds to the word count operation performed on the data file.

The exact format of the cache description of different applications varies according to their specific semantic contexts. This could be designed and implemented by application developers who are responsible for implementing their MapReduce tasks. In our prototype, we present several supported operations:

• **Item Count**. The count of all occurrences of each item in a text file. The items are separated by a user-defined separator.

• **Sort**. This operation sorts the records of the file. The comparison operator is defined on two items and returns the order of precedence.

• **Selection**. This operation selects an item that meets a given criterion. It could be an order in the list of items. A special selection operation involves selecting the median of a linear list of items.

• **Transform**. This operation transforms each item in the input file into a different item. The transformation is described further by the other information in the operation descriptions. This can only be specified by the application developers.

• **Classification**. This operation classifies the items in the input file into multiple groups. This could be an exact classification, where a deterministic classification criterion is applied sequentially on each item, or an approximate classification, where an iterative classification process is applied and the iteration count should be recorded.

Cache descriptions can be recursive. For example, in sequential processing, a data file could be processed by multiple worker nodes/processes. In that case, a cache item, generated by the final process, could be from the intermediate result files of a previous worker, so its description will be stacked together to form a recursive description. On the other hand, this recursive description could be expanded to an iterative one by directly appending the later operations to the older ones. However, this iterative description loses the context information about the later operations, that is, if another process is operating on a later cache item and is looking for potential cache that could save its own operations. By inspecting an iterative description, one cannot discern between a later cache item and a previous one because the origin of the cache item is the one that was fed by the application developers. In this way, the worker processes will be unable to precisely identify the correct cache item, even if the cache item is readily available.

2.2 Reduce phase cache description scheme

The input for the reduce phase is also a list of key-value pairs, where the value could be a list of values. Much like the scheme used for the map phase cache description, the original input and the applied operations are required. The original input is obtained by storing the intermediate results of the map phase in the DFS. The applied operations are identified by unique IDs that are specified by the user. The cached results, unlike those generated in the Map phase, cannot be directly used as the final output. This is because, in incremental processing, intermediate results generated in the Map phase are likely mixed in the shuffling phase, which causes a mismatch between the original input of the cache items and the newly generated input.

A remedy is to apply a finer description of the original input of the cache items in the reduce phase. The description should include the original data files generated in the Map phase. For example, two data files, “file1.data” and “file2.data”, are shuffled to produce two input files, “input1.data” and “input2.data”, for two reducers. “input1.data” and “input2.data” should include “file1.data” and “file2.data” as its shuffling source. As a result, new intermediate data files of the Map phase are generated during incremental processing; the shuffling input will be identified in a similar way. The reducers can identify new inputs from the shuffling sources by shuffling the newly-generated intermediate result from the Map phase to form the final results. For example, assume that "input3.data" is a newly generated results from Map phase; the shuffling results "file1.data" and "file2.data" include a new shuffling source, "input3.data". A reducer can identify the input "file1.data" as the result of shuffling "input1.data", "input2.data", and "input3.data". The final results of shuffling the output of "input1.data" and "input2.data" are obtained by querying the cache manager. The added shuffling output of "input3.data" is then added to get the new results.

Given the above description, the input given to the reducers is not cached exactly. Only a part of the input is identical to the input of the cache items. The rest is from the output of the incremental processing phase of the mappers. If a reducer could combine the cached partial results with the results obtained from the new inputs and substantially reduce the overall computation time, reducers should cache partial results. Actually, this property is determined by the operations executed by the reducers. Fortunately, almost all real-world applications have this property.

2.3 Case study with Hadoop MapReduce

2.3.1 Map cache

Apache Hadoop[2] (HDMR) is an open-source implementation of the MapReduce distributed parallel processing algorithm originally designed by Google. Map phase is a data-parallel processing procedure in which the input is split into multiple *file splits* which are then processed by an equal number of Map worker processes. As depicted in Fig. 3, a file split divides one or more input files based on user-supplied rules. The intermediate results obtained by processing file splits should be cached. Each file split is identified by the original *file name*, *offset*, and *size*. This identification scheme causes complications in describing cache items. In an ideal situation, each cache item would only correspond to a single input file, which makes identifying it with the above scheme straightforward. In reality, such a situation is seldom the case. This scheme is slightly modified to work for the general situation. The original field of a cache item is changed to a 3-tuple of {file_name, offset, size}. Note

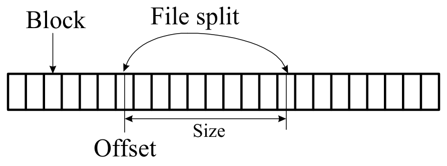


Fig. 3 A file in a DFS. This file is stored as multiple blocks, which are fixed-size data blocks. A file split is identified by the original file name, offset, and size.

that a file split cannot cross file boundaries in Hadoop MapReduce, which simplifies the description scheme of cache items. The operation field required by the cache description is described by a serialized Java object. This field is read by the Java program and tested against known Java class definitions to determine what operations are used.

Map cache items can be aggregated by grouping file splits. Multiple cache items that were generated from the same original file in the DFS are grouped under the path of the original file, i.e., {file_name, {offset, size}, {offset, size},$\cdots$}. The actual storage of aggregated cache items could be optimized accordingly. For example, they could be put on a single datanode in the HDFS cluster to avoid costly queries to multiple datanodes.

2.3.2 Reduce cache

Cache description in the reduce phase follows the designs in Section 2.2. The file splits from the map phase are included in the cache description. Usually, the input given to the reducers is from the whole input of the MapReduce job. Therefore, we could simplify the description by using the file name together with a version number to describe the original file to the reducers. The version number of the input file is used to distinguish incremental changes. A straightforward approach is to encode the size of the input file with the file name. Since we assume that only incremental changes, i.e., appending new data at the end of the file, are allowed, the size of the file is enough to identify the changes made during different MapReduce jobs. Note that even the entire output of the input files of a MapReduce job is used in the reduce phase, the file splits can still be aggregated as described in Section 2.1, i.e., by using the form of {file name, split, $\cdots$, split}.

As shown in Fig. 4, file splits are sorted and shuffled to generate the input for the reducers. Although this process is implicitly handled by the MapReduce framework, the users are able to specify a shuffling method by supplying a *partitioner*, which is

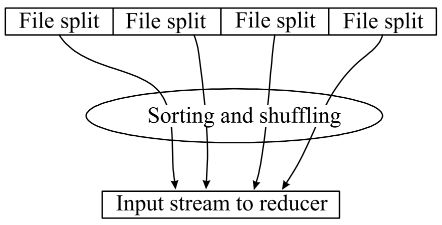


Fig. 4 The input stream to a reducer is obtained by sorting and then shuffling multiple output files of mappers. This mapping is used to identify the input to the reducer.

implemented as a Java object in Hadoop. The partitioner examines the key of a record and determines which reducer should process this record in the reduce phase. Therefore, the cache description should be attached with the partitioner, which can be implemented as a serialized object in Hadoop. The same input file splits that are partitioned by different partitioners produce different reduce inputs, therefore cannot be treated as the same. At last, the index of the reducer assigned by the partitioner is attached. The whole description is a 3-tuple: {file splits, partitioner, reducer index}. The description is completed to accurately identify the input to a reducer. The reducer then appends its output with the description to produce a cache item. However, This process is automatically handled by the reducers.

3 Protocol

3.1 Relationship between job types and cache organization

The partial results generated in the map and reduce phases can be utilized in different scenarios. There are two types of cache items: the map cache and the reduce cache. They have different complexities when it comes to sharing under different scenarios. Cache items in the map phase are easy to share because the operations applied are generally well-formed. When processing each file split, the cache manager reports the previous file splitting scheme used in its cache item. The new MapReduce job needs to split the files according to the same splitting scheme in order to utilize the cache items. However, if the new MapReduce job uses a different file splitting scheme, the map results cannot be used directly, unless the operations applied in the map phase are *context free*. By context free, we mean that the operation only generates results based on the input records, which does not consider the file split scheme. This is generally true.

When considering cache sharing in the reduce phase, we identify two general situations. The first is when the reducers complete different jobs from the cached reduce cache items of the previous MapReduce jobs, as shown in Fig. 5. In this case, after the mappers submit the results obtained from the cache items, the MapReduce framework uses the partitioner provided by the new MapReduce job to feed input to the reducers. The saved computation is obtained by removing the processing in the Map phase. Usually, new content is appended at the end of the input files, which requires additional mappers to process. However, this does not require additional processes other than those introduced above.

The second situation is when the reducers can actually take advantage of the previously-cached reduce cache items as illustrated in Fig. 6. Using the description scheme discussed in Section 2, the reducers determine how the output of the map phase is shuffled. The cache manager automatically identifies the best-matched cache item to feed each reducer, which is the one with the maximum overlap in the original input file in the Map phase.

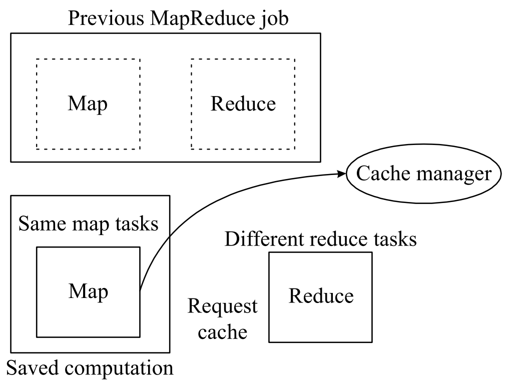


Fig. 5 The situation where two MapReduce jobs have the same map tasks, which could save a fraction of computation by requesting caches from the cache manager.

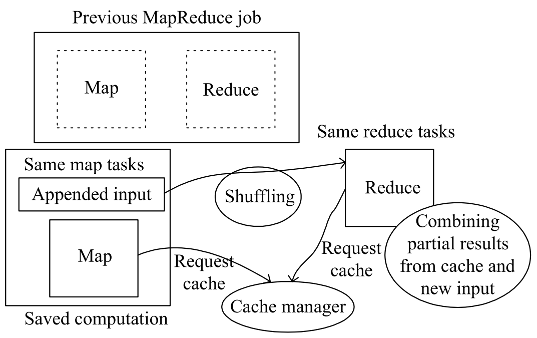


Fig. 6 The situation where two MapReduce jobs have the same map and reduce tasks. The reducers combine results from the cache items and the appended input to produce the final results.

3.2 Cache item submission

Mapper and reducer nodes/processes record cache items into their local storage space. When these operations are completed, the cache items are forwarded to the cache manager, which acts like a broker in the publish/subscribe paradigm[4]. The cache manager records the description and the file name of the cache item in the DFS. The cache item should be put on the same machine as the worker process that generates it. This requirement improves data locality. The cache manager maintains a copy of the mapping between the cache descriptions and the file names of the cache items in its main memory to accelerate queries. It also flushes the mapping file into the disk periodically to avoid permanently losing data.

A worker node/process contacts the cache manager each time before it begins processing an input data file. The worker process sends the file name and the operations that it plans to apply to the file to the cache manager. The cache manager receives this message and compares it with the stored mapping data. If there is a exact match to a cache item, i.e., its origin is the same as the file name of the request and its operations are the same as the proposed operations that will be performed on the data file, then the manager will send back a reply containing the tentative description of the cache item to the worker process.

The worker process receives the tentative description and fetches the cache item. For further processing, the worker needs to send the file to the next-stage worker processes. The mapper needs to inform the cache manager that it already processed the input file splits for this job. The cache manager then reports these results to the next phase reducers. If the reducers do not utilize the cache service, the output in the map phase could be directly shuffled to form the input for the reducers. Otherwise, a more complicated process is executed to obtain the required cache items, which will be explained in Section 3.4.

If the proposed operations are different from the cache items in the manager's records, there are situations where the origin of the cache item is the same as the requested file, and the operations of the cache item are a strict subset of the proposed operations. The concept of a *strict super set* refers to the fact that the item is obtained by applying some additional operations on the subset item. For example, an item count operation is a strict subset operation of an item count followed by a selection operation. This fact means that if we have a cache item for the first operation, we could just add the selection operation, which guarantees the correctness of the operation.

One of the benefits of Dache is that it automatically supports incremental processing. Incremental processing means that we have an input that is partially different or only has a small amount of additional data. To perform a previous operation on this new input data is troublesome in conventional MapReduce, because MapReduce does not provide the tools for readily expressing such incremental operations. Usually the operation needs to be performed again on the new input data, or the application developers need to manually cache the stored intermediate data and pick them up in the incremental processing. In Dache, this process is standardized and formalized. Application developers have the ability to express their intentions and operations by using cache description and to request intermediate results through the dispatching service of the cache manager.

3.3 Lifetime management of cache item

The cache manager needs to determine how much time a cache item can be kept in the DFS. Holding a cache item for an indefinite amount of time will waste storage space when there is no other MapReduce task utilizing the intermediate results of the cache item. There are two types of policies for determining the lifetime of a cache item, as listed below. The cache manager also can promote a cache item to a permanent file and store it in the DFS, which happens when the cache item is used as the final result of a MapReduce task. In this case, the lifetime of the cache item is no longer managed by the cache manager. The cache manager still maintains the mapping between cache descriptions and the actual storage location.

3.3.1 Fixed storage quota

Dache allocates a fixed amount of storage space for storing cache items. Old cache items need to be evicted when there is no enough storage space for storing new cache items. The eviction policy of old cache items can be modeled as a classic cache replacement problem[5]. In our preliminary implementation, the *Least Recent Used* (LRU) is employed. The cost of allocating a fixed storage quota could be determined by a pricing model that captures the monetary expense of using that amount of storage space. Such pricing models are available in a public Cloud service. We will discuss more details about the model in Section 3.3.2.

3.3.2 Optimal utility

Increasing the storage space of cache items will likely hit a plateau due to the diminishing return effect. A utility-based measurement can be used to determine an optimal space allocated for cache items which maximizes the benefits of Dache and respect the constraints of costs. This scheme estimates the saved computation time, t_s, by caching a cache item for a given amount of time, t_a. These two variables are used to derive the *monetary* gain and cost. The net profit, i.e., the difference of subtracting cost from gain, should be made positive. To accomplish this, an accurate pricing model of computational resources is required. Although conventional computing infrastructures do not offer such a model, cloud computing does. Monetary values of computational resources are well captured in existing cloud computing services, for example, in Amazon AWS[6] and Google Compute Engine[7]. For many organizations that rely on a cloud service provider for their IT infrastructure, this would be a perfect model. According to the official report from Amazon AWS, the amount of organizations that are actively using their services is huge, which help them to achieve near billion dollar revenue. Therefore, this pricing model should be very useful in real-world application. On the other hand, for organizations that rely on their own private IT infrastructure, this model will be inaccurate and should only be used as a reference.

$$\text{Expense}_{t_s} = P_{\text{storage}} \times S_{\text{cache}} \times t_s \tag{1}$$

$$\text{Save}_{t_s} = P_{\text{computation}} \times R_{\text{duplicate}} \times t_s \tag{2}$$

Equations (1) and (2) show how to compute the expense of storing cache and the corresponding saved expense in computation. The details of computing the variables introduced above are as follows. The gain of storing a cache item for t_s amount of time is calculated by accumulating the charged expenses of all the saved computation tasks in t_s. The number of the same task that is submitted by the user in t_s is approximated by an exponential distribution. The mean of this exponential distribution is obtained by sampling in history. A newly-generated cache item requires a bootstrap time to do the sampling. The cost is directly computed from the charge expense of storing the item for t_a amount of time. The optimal lifetime of a cache item is the maximum t_a, such that the profit is positive. The overall benefits of this scheme are that the user will not be charged more and at the same time the computation time is reduced, which in turn reduces the response time and increases the user satisfaction.

3.4 Cache request and reply

3.4.1 Map cache

There are several complications that are caused by the actual designs of the Hadoop MapReduce framework. The first is, when do mappers issue cache requests? As described above, map cache items are identified by the data chunk and operations performed. In order to preserve the original splitting scheme, cache requests must be sent out before the file splitting phase. The jobtracker, which is the central controller that manages a MapReduce job, issues cache requests to the cache manager. The cache manager replies a list of cache descriptions. The jobtracker then splits the input file on remaining file sections that have no corresponding results in the cache items. That is, the jobtracker needs to use the same file split scheme as the one used in the cache items in order to actually utilize them. In this scenario, the new appended input file should be split among the same number of mapper tasks, so that it will not slow the entire MapReduce job down. Their results are then combined together to form an aggregated Map cache item. This could be done by a nested MapReduce job.

3.4.2 Reduce cache

The cache request process is more complicated. The first step is to compare the requested cache item with the cached items in the cache manager's database. As described in Section 2.2, the cached results in the reduce phase may not be directly used due to the incremental changes. As a result, the cache manager needs to identify the overlaps of the original input files of the requested cache and stored cache. In our preliminary implementation, this is done by performing a linear scan of the stored cache items to find the one with the maximum overlap with the request. When comparing the request and cache item, the cache manager first identifies the partitioner. The partitioner in the request and the cache item have to be identical, i.e., they should use the same partitioning algorithm and the same number of reducers. This requirement is illustrated in Fig. 7. The overlapped part means that a part of the processing in the reducer could be saved by obtaining the cached results for that part of the input. The incremented part, however, will need to be processed by the reducer itself. The final results are generated by

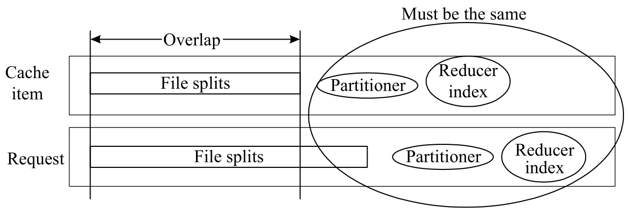


Fig. 7 In order to compare a cache description and a cache request, the cache manager needs to examine the partitioner and the reducer indexes and make sure that they are the same.

combining both parts. The actual method of combining results is determined by the user.

4 Performance Evaluation

4.1 Implementation

We extend Hadoop to implement Dache. Hadoop is a collection of libraries and tools for DFS and MapReduce computing. The complexity of the entire package is beyond our control, so we take a non-intrusive approach to implement Dache in Hadoop and try not to hack the Hadoop framework itself, but implement Dache by changing the components that are open to application developers. Basically, the cache manager is implemented as an independent server. It communicates with task trackers and provides cache items on receiving requests. The cache manager stands outside of the Hadoop MapReduce framework. The cache manager uses HDFS, the DFS component of Hadoop, to manage the storage of cache items.

In order to access cache items, the mapper and reducer tasks first send requests to the cache manager. However, this cannot be implemented in Mapper and Reducer classes. Hadoop framework fixes the interface of Mapper and Reducer classes to only accept key-value pairs as the input. They cannot identify the file split they are working on; therefore, cache requests cannot be sent from mappers or reducers. We alter two components of Hadoop to implement this function. The first component is InputFormat class, an open-accessed component that allows application developers to modify. It is responsible for splitting the input files of the MapReduce job to multiple file splits and parse data to key-value pairs. As stated in Section 2.1, InputFormat class should query the cache manager to fetch the splitting scheme of the cache item, if they are the same Map tasks that were being executed previously. It then splits the input files in the same way as the cache item and puts the incremental parts into new file splits. The second component that needs to be altered is the TaskTracker, which is the class responsible for managing tasks. TaskTracker is able to understand filesplit and bypass the execution of mapper classes entirely. TaskTracker also manages reducer tasks. Similarly, it could bypass reducer tasks by utilizing the cached results. Additionally, application developers must implement a different reduce interface, which takes as input a cache item and a list of key-value pairs and produces the final results.

Our non-intrusive approach has a performance penalty. It is because Dache fundamentally requires changes to the MapReduce framework to better utilize cache items. For example, InputFormat incurs an overhead in following the splitting scheme of the cache items. This could be avoided by employing a native cache query mechanism by altering the MapReduce framework.

4.2 Experiment settings

Hadoop is run in pseudo-distributed mode on a server that has an 8-core CPU, each core running at 3 GHz, 16 GB memory, and a SATA disk. The number of mappers is 16 in all experiments, the reducers' count varies. We use two applications to benchmark the speedup of Dache over Hadoop (the classic MapReduce model): *word-count* and *tera-sort*. Word-count counts the number of unique words in large input text files; tera-sort sorts key-value records based on the lexical order of the key. More details are in Hadoop manual[2]. Word-count is an IO-intensive application that requires loading and storing a sizeable amount of data during the processing. On the other hand, tera-sort uses more mixed word loads. It needs to load and store all input data and needs a computation-intensive sorting phase. The inputs of two applications are generated randomly, and all are 10 GB in size.

4.3 Results

Figures 8 and 9 present the speedup and completion time of two programs. The completion time and the speedup are put together. Data is appended to the input file. The size of the appended data varies and is represented as a percentage number to the original input file size, which is 10 GB. Tera-sort is more CPU-bound compared to word-count, as a result Dache can bypass computation tasks that take more time, which achieves larger speedups. The speedup decreases

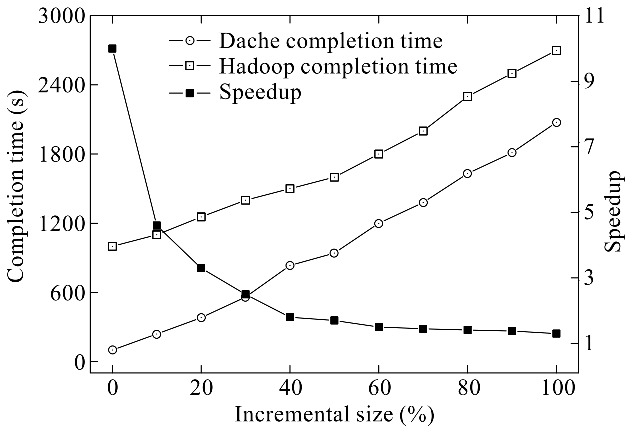


Fig. 8 The speedup of Dache over Hadoop and their completion time of word-count program.

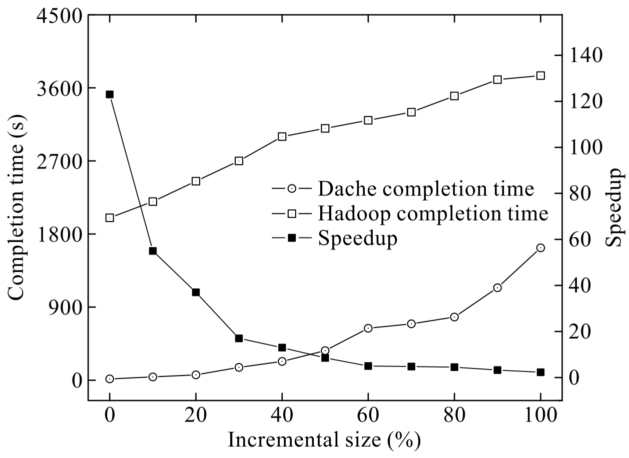


Fig. 9 The speedup of Dache over Hadoop and their completion time of tera-sort program.

with the growing size of appended data, but Dache is able to complete jobs faster than Hadoop in all situations. The map phase of tera-sort does not perform much computation, which also makes it easier for Dache to work.

Figures 10 and 11 show the CPU utilization ratio of the two programs. It is measured by averaging the CPU utilization ratio of the processes of the MapReduce jobs over time. Tera-sort consumes more CPU cycles than word-count does, which is determined by the CPU-bound nature of the sorting procedure. From the figures, it is clear that Dache saves a significant amount of CPU cycles, which is demonstrated by the much lower CPU utilization ratio. These results are consistent with Figs. 8 and 9. With a larger incremental size, the CPU utilization ratio of Dache grows significantly, too. This is because Dache needs to process the new data and cannot utilize any cached results for bypassing computation tasks. Figures 8-11 collectively prove that Dache indeed removes redundant tasks in incremental MapReduce jobs and reduces job completion time.

Figure 12 presents the size of all the cache items produced by a fresh run of the two programs with different input data sizes. In tera-sort, cache items should have the same size as the original input data because sorting does not remove any data from the input. The difference between the input data size and the cache size is caused by the data compression. Note also that the cache item in tera-sort is really the final output,

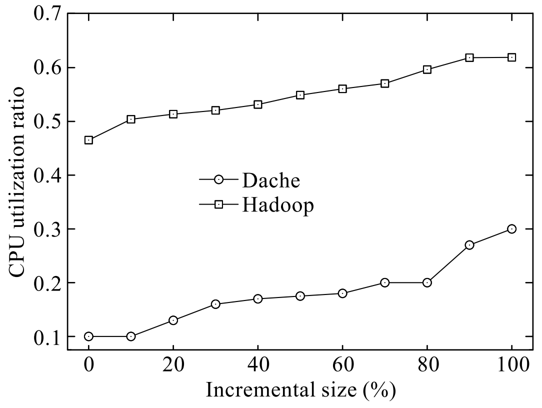


Fig. 10 CPU utilization ratio of Hadoop and Dache in the word-count program.

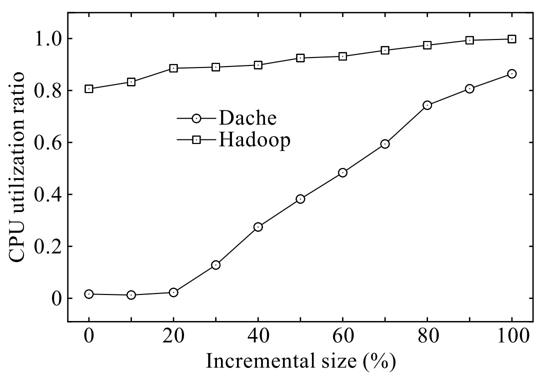


Fig. 11 CPU utilization ratio of Hadoop and Dache in the tera-sort program.

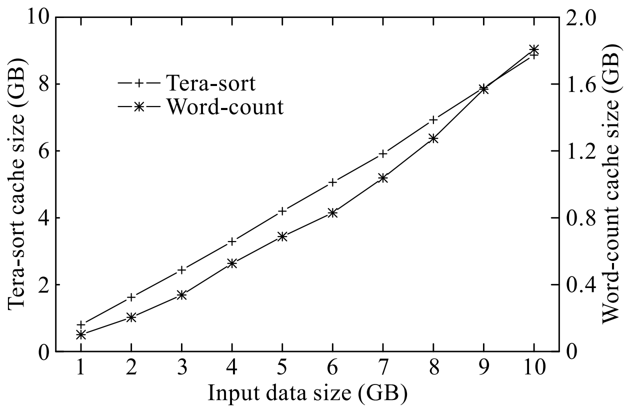


Fig. 12 Total cache size in GB of two programs.

which means that the used space is free in the sense that no extra cost is incurred in storing cache items. The word-count results are more related to the input record distribution.

5 Related Work

A comprehensive survey on incremental computation is given in Ref. [8]. Google Bigtable[9] is a distributed storage system for managing structured data built on top of the Google File System (GFS)[10]. It is able to efficiently handle incremental processing with the structure information in the data. Google Percolator[11] is an incremental processing platform, which achieves much faster web index processing compared to the previous MapReduce-like batch-processing system. It is used in Google's new web page indexing engine, which achieves 50% fresher web indexing than the old system. Ramcloud[12] is a distributed computing platform, where all data is kept in RAM instead of on disk. Unlike Hadoop, Ramcloud focuses on the computation and processing performed on small data objects.

Dryad[13], is a distributed programming model that is targeted at the same application scenarios as MapReduce. Unlike MapReduce's simple two-phase execution model, Dryad employs a Directed Acyclic Graph (DAG) based model. Dryad is thus able to provide a more natural representation of many real-world problems. However, from an implementation point of view, such a design causes excessive complexity for application developers to implementation with Dryad, which substantially hinders its adoption. DryadInc[14] is an extension to Dryad to reuse identical work to accelerate processing. NOVA[15] is a workflow manager proposed designed for the incrementally executing Pig[16] programs upon streaming data. CEAL[17,18] is a framework for dynamically describing application computation requirements in a cloud environment. Our work focuses on dynamically identify redundant computation in MapReduce job.

Memorycached[19] is a distributed caching system designed as an object accessing layer between the application and underlying relational database. The cache manager of Dache could utilize Memorycached to accelerate query response because the size of cache item is generally small. Scheuermann et al.[20] studied how to optimize the performance of distributed storage with disks. In Ref. [21], Zaharia et al.[21] studied the speculative execution of tasks that potentially slows down the entire MapReduce job in order to accelerate the execution of a MapReduce job. This work does not address the data sharing problem identified in this paper. This mechanism is orthogonal to ours and could be integrated straightforwardly.

Performance optimization in data-intensive applications with MapReduce is an active research topic. Herodotou et al.[22] proposed a intelligent cluster sizing algorithm for data-intensive analytics applications. Wu et al.[23] studied the query optimization problem in using MapReduce to do online query processing. Both Wu et al.'s work and our work aim to accelerate processing by removing redundancy in the computing process. Their work is focused on a higher layer than ours. Logothetis et al.[24] presented a general architecture for continuous bulk processing. Although it has a similar problem as ours, it focuses on programming models instead of actual implementation. Comet[25] and Nephele[26] are frameworks for batch analytical processing on bulk data. The input data is modeled as a stream. The system answers query by utilizing the continuing coming data stream.

ActiveSLA[27] is an admission control framework that takes into account the monetary gain of admitting jobs to run on a cloud-based database service provider. Their methodology is similar to ours in using actual profit to make management decisions. This work could be refined by taking into account the saved costs by employing Dache. Google fushion tables[28] is a service for integrating data in Cloud platform, which has integrated data analytic capability.

6 Conclusions

We present the design and evaluation of a data-aware cache framework that requires minimum change to the original MapReduce programming model for provisioning incremental processing for Big-data applications using the MapReduce model. We propose Dache, a data-aware cache description scheme, protocol, and architecture. Our method requires only a slight modification in the input format processing and task management of the MapReduce framework. As a result, application code only requires slight changes in order to utlize Dache. We implement Dache in Hadoop by extending relevant components. Testbed experiments show that it can eliminate all the duplicate tasks in incremental MapReduce jobs and does not require substantial changes to the application code. In the future, we plan to adapt our framework to more general application scenarios and implement the scheme in the Hadoop project.

Acknowledgements

This research was supported in part by the Natural Science Foundation of USA (Nos. ECCS 1128209, CNS 1138963, CNS 1065444, and CCF 1028167).

References

[1] J. Dean and S. Ghemawat, Mapreduce: Simplified data processing on large clusters, *Commun. of ACM*, vol. 51, no. 1, pp. 107-113, 2008.

[2] Hadoop, http://hadoop.apache.org/, 2013.

[3] Java programming language, http://www.java.com/, 2013.

[4] P. Th. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, The many faces of publish/subscribe, *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114-131, 2003.

[5] Cache algorithms, http://en.wikipedia.org/wiki/Cache algorithms, 2013.

[6] Amawon web services, http://aws.amazon.com/, 2013.

[7] Google compute engine, http://cloud.google.com/ products/computeengine.html, 2013.

[8] G. Ramalingam and T. Reps. A categorized bibliography on incremental computation, in *Proc. of POPL '93*, New York, NY, USA, 1993.

[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data, in *Proc. of OSDI'2006*, Berkeley, CA, USA, 2006.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung, The google file system, *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29-43, 2003.

[11] D. Peng and F. Dabek, Largescale incremental processing using distributed transactions and notifications, in *Proc. of OSDI' 2010*, Berkeley, CA, USA, 2010.

[12] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazi'eres, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, The case for ramcloud, *Commun. of ACM*, vol. 54, no. 7, pp. 121-130, 2011.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, Dryad: Distributed data-parallel programs from sequential building blocks, *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 59-72, 2007.

[14] L. Popa, M. Budiu, Y. Yu, and M. Isard, Dryadinc: Reusing work in large-scale computations, in *Proc. of HotCloud'09*, Berkeley, CA, USA, 2009.

[15] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. N. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang, Nova: Continuous pig/hadoop workflows, in *Proc. of SIGMOD'2011*, New York, NY, USA, 2011.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, Pig latin: A not-so-foreign language for data processing, in *Proc. of SIGMOD'2008*, New York, NY, USA, 2008.

[17] U. A. Acar, Self-adjusting computation: An Overview, in *Proc. of PEPM'09*, New York, NY, USA, 2009.

[18] T. Karagiannis, C. Gkantsidis, D. Narayanan, and A. Rowstron, Hermes: Clustering users in large-scale e-mail services, in *Proc. of SoCC '10*, New York, NY, USA, 2010.

[19] Memcached—A distributed memory object caching system, http://memcached.org/, 2013.

[20] P. Scheuermann, G. Weikum, and P. Zabback, Data partitioning and load balancing in parallel disk systems, *The VLDB Journal*, vol. 7, no. 1, pp. 48-66, 1998.

[21] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, Improving mapreduce performance in heterogeneous environments, in *Proc. of OSDI'2008*, Berkeley, CA, USA, 2008.

[22] H. Herodotou, F. Dong, and S. Babu, No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics, in *Proc. of SOCC'2011*, New York, NY, USA, 2011.

[23] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi, Query optimization for massively parallel data processing, in *Proc. of SOCC'2011*, New York, NY, USA, 2011.

[24] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum, Stateful bulk processing for incremental analytics, in *Proc. of SOCC'2011*, New York, NY, USA, 2010.

[25] B. He, M. Yang, Z. Guo, R. Chen, B. Su, W. Lin, and L. Zhou, Comet: Batched stream processing for data intensive distributed computing, in *Proc. of SOCC'2011*, New York, NY, USA, 2010.

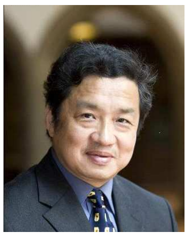
[26] D. Battr'e, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, Nephele/pacts: A programming model and execution framework for web-scale analytical processing, in *Proc. of SOCC'2010*, New York, NY, USA, 2010.

[27] P. Xiong, Y. Chi, S. Zhu, J. Tatemura, C. Pu, and H. HacigümüŞ, Activesla: A profit-oriented admission control framework for database-as-a-service providers, in *Proc. of SOCC'2011*, New York, NY, USA, 2011.

[28] H. Gonzalez, A. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, and W. Shen, Google fusion tables: Data management, integration and collaboration in the cloud, in *Proc. of SOCC'2010*, New York, NY, USA, 2010.

Yaxiong Zhao is a software engineer at Google's Platforms Networking group. He was with Amazon Web Services before joining Google, working on Amazon Kinesis, the first hosted stream data processing service in the world. He received his PhD degree from Temple University, USA. His research interests are in distributed systems, cloud computing, and wireless networks. He was the founding chair of the First International Workshop on Resource Management of Cloud Computing. His current focus is on software defined networking and data center networking for cloud computing.

Cong Liu is currently an assistant professor at Sun Yat-sen (Zhongshan) University, China. He received his PhD degree from Florida Atlantic University, USA in 2009. Before that he received his MS degree in computer software & theory from Sun Yat-sen (Zhongshan) University, China, in 2005. He received his BS degree in micro-electronics from South China University of Technology in 2002. He has served as a reviewer for many conference and journal papers. His main research interests include routing in Delay Tolerant Networks (DTNs) routing, geometric routing in Mobile Ad hoc Networks (MANETs), deep packet inspection, transaction processing, and rough set theory.



Jie Wu is the chair and a Laura H. Carnell Professor in the Department of Computer and Information Sciences at Temple University. Prior to joining Temple University, USA, he was a program director at the National Science Foundation and a distinguished professor at Florida Atlantic University. He received his PhD degree from Florida Atlantic University in 1989. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. Dr. Wu regularly published in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including IEEE Transactions on Computers, IEEE Transactions on Service Computing, and Journal of Parallel and Distributed Computing. Dr. Wu was general co-chair/chair for IEEE MASS 2006 and IEEE IPDPS 2008 and program co-chair for IEEE INFOCOM 2011. Currently, he is serving as general chair for IEEE ICDCS 2013 and ACM MobiHoc 2014, and program chair for CCF CNCC 2013. He was an IEEE Computer Society Distinguished Visitor, ACM Distinguished Speaker, and chair for the IEEE Technical Committee on Distributed Processing (TCDP). Dr. Wu is a CCF Distinguished Speaker and a Fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award.